Functional Databases, Functional Languages

Computation Structures Group Memo 259
14 November 1985

Rishiyur S. Nikhil

Funding for the Laboratory for Computer Science is provided by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-83-K-1025.

Functional Databases, Functional Languages

Rishiyur S. Nikhil
MIT Laboratory for Computer Science[1]
545 Technology Square
Cambridge, MA 02139
USA

Abstract

Database systems today have evolved a great deal from the first storage structures. This evolution has been towards greater data independence, expressive power in manipulation languages (for query and update), and expressive power in data models. But they are still poor substitutes for the much better understood analogues in programming languages: data abstraction, structured language constructs and type systems respectively. Programming languages, on the other hand, deal inadequately (if at all) with the question of long-lived structured data. The problem is compounded when one has to deal with both a database system and a programming language that are alien to each other (as in most "applications programming" today).

We believe that functional programming languages with functional databases offer a clean solution to this problem. Functional databases are databases that are never updated--rather, one views them as infinite sequences of "versions". Functional programming languages allow functions as data objects and do not have any "update" operation.

We outline the many advantages that such a system has to offer, pertaining to types, optimization, expressive power, and interactive environments.

[1]Funding for the Laboratory for Computer Science is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-83-K-1025.

1. What's Wrong with Databases Today

Database systems have evolved out of mass-storage device architectures. This evolution has moved away steadily from specific device architectures to more and more abstract models-- for example, relational databases. Even so, they are difficult to use and most applications do so only when efficient and reliable access to large amounts of data is critical. Hundreds of everyday applications (mail systems, software engineering aids, word processors, ...) prefer *not* to deal with the complexities of current database systems and make do with the much more primitive file systems that are commonly available today.

Let us examine some of the charecteristics that make current database systems so difficult and inconvenient to use.

1.1. Inadequate Type Structure

To a first approximation, the term "Data Model" used in the database community is analogous to the "Type Theory" of a programming language, *i.e.* it specifies the universe of definable types. A "schema" for a database is analogous to a particular type definition within that type theory, and a "database" conforming to a schema is analogous to a particular value or data structure that has that type.

The type-structures of most database systems are extremely poor in comparison to those available in modern programming languages. For example [Date,C.J. 75], CODASYL databases provide scalar types, tuples (*i.e.* records) of scalars, sequences of records[2], and record-sequence pairs (owner and members of a set), together with various means to share records amongst different sequences. Relational databases provide scalar types, and sets of tuples containing scalar types (*i.e.* relations).

Unfortunately, these structuring mechanisms are not suitable for many applications. The encoding of data from the problem domain into (say) relations may be extremely awkward and is thus error-prone. For example, parse trees, graphs, intermediate code, IC layouts, matrices, nested sets, ... are difficult to represent with relations. Forcing a programmer to use only relations adds to his intellectual burden and is reminiscent of the days when one had to encode complicated structures such as graphs and trees in FORTRAN arrays.

Some of the newer so-called "semantic data models" attempt to remedy this

[2]Unfortunately called "sets" in CODASYL terminology.

shortcoming [Lochovsky,F.H. 82]. But we feel that they are still poor substitutes for the type facilities of modern programming languages.

Most current data models permit the description only of "passive" collections of data-- they do not integrate data-manipulation *procedures* as part of the schema for a particular database. Instead, there is typically a single, pre-determined generic set of database operators. This separation leads to two problems: it is difficult smoothly to mix intensionally- and extensionally-defined data, and it does not allow data abstraction.

The former capability is useful, for example, to let the implementation of an abstract type choose freely between actually storing a table (such as a table of thermodynamic coefficients) versus regenerating the entries of the table on demand. Depending on space and time requirements, one or the other, or even a combination, may be preferable.

We shall have more to say about data abstraction capabilities in the next section.

A serious shortcoming of most data models is that they are rarely accompanied by rigorous semantics. Even in the relational model, which is one of the cleanest, there are many fuzzy areas, including the meanings of duplicate column names, association of a single column name with different domains in different relations, grouping of tuples within relations, the exact meaning of attribute domains, various kinds of "join" operations, etc.

1.2. Lack of Data Abstraction

The lack of data abstraction facilities is another serious problem in most data models. In the database world, one of the goals has long been to achieve "data independence" [Date,C.J. 75], *i.e.* the independence of application programs from the particular access methods and storage structures used to implement a conceptual schema. Current database systems achieve this to varying degrees.

But this concept is well understood in the programming language world. It is in fact nothing other than the "representation independence" of abstract data types, and is in fact exactly what makes these types "abstract". It is well known how to achieve this-- each abstract type is be accompanied by a set of procedures to manipulate objects of that type, and one uses linguistic mechanisms to ensure that the *only* way to manipulate objects of an abstract type is to use one of those associated procedures. If the representation of the abstract type is changed, one need only change the implementations of the associated procedures; all client applications will be unaware of and unaffected by the change [Liskov,B.H. 74].

It is often necessary to maintain a notion of "consistency" in a database beyond what it is expressible in the type system (for example, that the standard deviation of employee salaries should not exceed a certain number). Updates that result in inconsistent states of the database should not be permitted. Definitions of consistency are usually application-specific and may be arbitrarily complex.

Many data models rely on a separate language of "integrity constraints" to express consistency requirements. There are several problems with this approach. First, it adds yet another language for the programmer to understand and master. Second, because the constraint language is not integrated with the data manipulation language, it is difficult to implement the checking of constraints by other than the naive (and inefficient) method of performing the entire update, checking the consistency criterion from scratch, and, if it is found to be violated, undoing the update. Third, because of this separation of the constraint and data manipulation languages, the programmer is not left with much flexibility in error-handling-- *i.e.* to trap such violations, identify fine-grained reasons for the violation, and to take corrective action. For these and other reasons, languages for integrity constraints are often deliberately simplistic, incapable of expressing complex consistency requirements.

In the programming language world, one again uses abstract data types to maintain integrity constraints (otherwise known as "invariants"). Abstract data types ensure that data of a particular type can be manipulated only by the procedures associated with that type. These procedures are specified by the programmer and he has the full power of the programming language available to express integrity checks as complicated as the application demands, with whatever efficiency is possible, and with whatever error-handling he wishes. In the above example, the programmer may choose to maintain the standard deviation of employee salaries internal to the employee abstract type, and recompute it incrementally on each update; this is much more efficient than recomputing it from scratch on each update.

"Alerters" are another useful feature of databases [Buneman 77]. Like integrity constraints, an alerter monitors the database for some condition, except that these are not necessarily violations of consistency-- e.g. large monetary withdrawal, fuel-level low, etc. On such conditions, an alerter performs an associated action, such as sending a warning message to a responsible person. Again, like integrity constraints, it is much better to be able to incorporate alerters into the procedure definitions in an abstract data type rather than use a separate alerter language and a separate implementation technique.

1.3. Lack of Integration with Programming Languages

Most databases can be queried and updated in two ways: by using a stand-alone query/update language (e.g. SQL, QUEL, ...), or by embedding a database "sublanguage" in an existing programming language (e.g. EQUEL in C, SQL in PL/1, ...) [Date,C.J. 75].

Stand-alone database languages are generally convenient to use, but are typically very limited in their computational power. They usually have the generic operators for manipulating the data-structures supported by the data model, and may include some "library functions" that perform a few other common computations (e.g. minimums and maximums, totals, averages). But for more complicated or non-standard operations, one must resort to the other method.

The use of a database sublanguage embedded in a regular programming language is notoriously difficult, due to the usually great semantic mismatch between the two. Typically, the type structure of the programming language is entirely different from that of the database system. For example, the database may be relational, but there may be no type in the programming language that has a natural correspondence with relations. Further, the programming language generally has no control structures that may be used naturally with the database structures.

The methods used to circumvent this mismatch are usually cumbersome and unsafe. For example a procedure may send a message to be interpreted by the database system, which leaves a result in some invisible work area. To examine this result subsequently, it may be necessary to establish a "cursor" on it, which must then be moved in small sequential increments revealing a small piece of the result at a time. What is worse, this kind of interface may relinquish completely any type-checking that the programming language normally provides. The imposition of such unnatural control structures to traverse database objects can negate any data-independence that the data model alone originally provides. The programmer must now carry the additional intellectual burden of managing two dissimilar models of memory: that of the programming language itself, and that of the database system.

2. What Database Features do Programming Languages Lack?

We have claimed that many database concepts have better analogues in programming languages. What then are the particular characteristics of databases that programming languages lack?

1. Databases are not *ephemeral*, *i.e.* databases are data-structures that are

"permanent". In contrast, most data-structures in programs have a lifetime no longer than that of the program itself.

2. Databases address the problem of efficient access to data structures implemented in a memory heirarchy that includes devices with widely varying charecteristics such as latency, bandwidth and size of the unit of transfer.

3. Databases are resilient. Data are remembered in spite of hardware failures.

4. Databases provide facilities such as access control, security, etc.

(Not all databases today provide all these features!)

Database system designers have always attached great importance to these characteristics while slowly evolving databases away from specific device architectures and incorporating more and more features normally found in programming languages.

3. What Should a Database Programming Language Have?

We believe that the bottom-up evolution of database systems outlined in the previous section can lead to some short-sighted views of what databases are and what they should be. We have often experienced the following situation: when we point out the poverty of today's data models in comparison with programming-language type systems, database researchers often respond that such richness and generality is "just not necessary in database work". In our opinion, this is patently false. It is just a symptom of the unfortunate fact that the requirements of "database work" tend to get defined in terms of what is currently available. We firmly believe that if richer type systems were available in databases, we would not want for customers.

We therefore prefer to explore the opposite evolutionary path: that of beginning with a programming language, attaching great importance to the expressive power and semantic elegance of the language, while gradually incorporating features normally found only in databases, putting to good use all the lessons learned in implementing today's database systems.

Having explored the relative strengths of programming languages and databases, let us look at some the features we would like to have in an integrated system.

Our primary requirement for a database programming language is a rich type system in which to model naturally structures in a wide variety of application domains. Objects in this type system should directly *be* the database, *i.e.* there should be no distinction between programming language types and database types. Such a type system should have user-defined types and abstract data types. In addition, it should support the subtyping and inheritence mechanisms typical of Simula and Smalltalk. The type system should have rigourous semantics.

Along with a rich type system, it is essential for the language to permit functions as values, for two reasons. First, it is convenient to be able to store functions as values in the database (e.g. a tax-computation function customized for each employee). Second, for each new type that the programmer defines, he usually wishes it to be accompanied by control-structures convenient for manipulating objects of that type. This is a powerful abstraction mechanism that depends on the ability to treat functions as values.

There have been some attempts to integrate programming languages with databases. For example, Pascal/R [Schmidt,J.W. 77] is a system that integrates a relational database system with Pascal. It extends the type system of Pascal to include the **relation** type built on Pascal records, and the **database** type built on relations, and it extends the control structures of Pascal to include constructs to traverse and manipulate relations. In our opinion this is a very successful and elegant integration of a conventional database and programming language[3].

There are also many exciting efforts currently under way to integrate logic programming languages (such as Prolog) with relational database systems to create so-called "knowledge bases". The concept exploited here is that ground "facts" in a logic program have a natural interpretation as a relation in a relational database. This approach thus gives relational databases a "reasoning power", and allows the smooth integration of intensionally- and extensionally-defined data.

However, we believe that neither of these approaches meet our objectives. The languages used are not expressive enough in their own right (no functional values, no data abstraction, inadequate facilities for modularity). And, in addition, the type structures provided for persistent objects are too limited (just relations, in both cases).

PS-Algol [Atkinson,M.P. 81] is an extension of S-Algol (which is itself derived from Algol60) to permit the persistence of arbitrarily typed objects. A database exists external to the program, and there are mechanisms to make the database appear as part of the standard program heap. Objects in the heap may be modified using normal language constructs, and any objects on the heap may be made to persist beyond the lifetime of the

[3]This research has since evolved into the design of DBPL, a relational programming language based on Modula-2, reported elsewhere in these proceedings.

program by entering them directly or indirectly into distinguished tables at the top-level of the program. In PS-Algol, type-checking is not very strong because pointers are untyped, and because there is no abstract data type facility[4].

Galileo [Albano,A. 83] is a database programming language with a rich type system based on that of ML [Albano,A. 85a, Cardelli,L. 83]. In Galileo, the database is part of a global environment maintained by a host system. The database may contain updatable objects. A database program is an expression evaluated in this environment, and this evaluation may modify the updatable objects in the database [Albano,A. 85b].

Both these systems modify the database by side-effect. In contrast, we are exploring a database programming language with an essential difference: the database is *immutable*, *i.e.* it is a "Functional Databases". An "update transaction" conceptually produces a new version of the database. The languages used both for querying and "updating" this database are functional languages-- languages in which there are no side effects, and in which functions are values just like any other.

We call this class of database programming languages "FDBPLs", or Functional Database Programming Languages. We explicate and justify this approach in the next section, but first we present our view of the evolution of DBPLs in Figure 3-1.

4. Functional Database Programming Languages

In this section we outline our view of databases in a functional setting for query and update, and show the numerous advantages it has to offer.

4.1. What is a Database in an FDBPL?

We view a database as an *environment* of type and value bindings (i.e. a mapping from identifiers to types and values). Note that values can be arbitrarily complex (scalars, arrays, relations, functions, streams, values of abstract types, ...).

A *Query* in such a database is merely an expression evaluated in that environment, producing a value that is the "answer".

For example, the following is an environment representing a student-course database:

[4]This research has since evolved into the design of Napier, a persistent language with richer types and stronger type-checking [Atkinson,M.P. 85].

File Structures + Conventional Programming Language	---	COBOL, PL/1 ...
⇓		
DB with limited types + alien PLs	--- ---	Most current DBMSs
⇓		
Integrated DBPL with limited types	--- ---	Pascal/R, DBPL, Prolog + RDB
⇓		
Integrated DBPL with richer types	---	PS-Algol
⇓		
Integrated DBPL with very rich types	--- ---	Galileo, Napier, FDBPL

Figure 3-1: Evolution of Database Programming Languages

```
type Student
value Students: list(Student)
value Age:   Student -> Number
value Sname: Student -> String

type Course
value Courses: list(Course)
value Cname: Course -> String
value Core:  Course -> Boolean

value TakenBy: Course  -> list(Student)
value Takes:   Student -> list(Course)
```

A query to find the names of all students older than 30 taking core courses may be written:

```
let
  coreCourses  = filter Core Courses ;
  coreStudents = flatten (map TakenBy coreCourses) ;
  oldStudent s = (age s) > 30
in
  filter oldStudent coreStudents
```

where `filter p s` returns the subsequence of `s` containing just those members of `s` that satisfy the predicate `p`, `map f s` returns a sequence containing the result of applying the function `f` to each member of the sequence `s`, and `flatten ss` returns a sequence containing all members of the sequence of sequences `ss`. This view of functional query languages and methods to implement them efficiently are explored in detail in [Nikhil,R.S. 84].

In a functional database, queries and updates are treated as activities at two separate levels. At the query level, the user supplies an expression to be evaluated within an environment. The database *is* the environment in which the expression is evaluated, and the resulting value is the answer to the query[5]. Updates on the other hand occur at the *meta-level* with respect to queries-- here one evaluates an expression that treats the database (or environment) as an object, and produces a new database. The applicative language framework used at both levels is identical; it is only the objects being manipulated that are different.

A query or update to the database, by default, is always made with respect to the "latest" version. But this can be generalized quite easily so that subexpressions may be syntactically qualified explicitly by the environment in which they are to be evaluated, thus allowing one to write queries and updates that depend not only on the "latest" version of the database but also on previous versions.

4.2. Why Functional Languages?

Expressiveness: That functional languages have great elegance and expressive power has been recognized for some time [Backus 78, Burge 75, Turner,D.A. 81]. Various high-level features of functional languages give them this expressive power.

In functional languages, functions are first-class values. This is a powerful abstraction mechanism that permits *the programmer* to design appropriate control structures for existing and new types of objects, thus leading to very compact, transparent notation. For example, in defining a new type of table in a database, one can simultaneously define general-purpose generators, iterators and reducers to operate on such tables, so that subsequent uses of such tables are concise and clear. Burge [Burge 75] gives numerous programming examples that demonstrate the power of this facility.

Lazy evaluation is a powerful intellectual tool that makes it not only feasible, but even efficient to define and manipulate infinite and/or large objects [Turner,D.A. 81]. In addition, lazy evaluation has certain automatic database optimization capabilities in that unnecessary computations (which may involve disk accesses) never get done [Nikhil,R.S. 84]. To make

[5]Note that the "value" computed by a query may be simple-- such as a number representing a salary-- or as complex as one wishes, such as the sequence of students in the above example.

effective use of lazy evaluation, however, we need a functional language; lazy evaluation does *not* interact well with updates. The technique of lazy evaluation depends on a lack of constraints on when a particular expression needs to be evaluated, so that it may be safely postponed. Thus with lazy evaluation, the order in which expressions are evaluated, and indeed the question of whether an expression is evaluated at all, may be quite unpredictable. If such expressions contain side-effects, the behaviour of programs can be extremely surprising.

Functional languages are expression-based-- the main composition rule is function application and is used uniformly from small expressions to large programs. Thus functional languages are very suitable for uniform interactive environments suitable not only for quick, small, one-off queries but also for the development of large compiled programs.

Even though we believe that functional languages are more "user-friendly" because of their clean semantics and high-level features, they are still formal languages, and under certain circumstances (e.g. casual use of a database by untrained users) one may wish to use other front-ends such as natural-language or graphical interfaces. Even in such situations, we believe that because of their regularity, functional languages make ideal *target* languages into which queries in, say, English are first compiled.

Opportunities for query optimization via program transformation: The semantic cleanliness of functional languages makes it feasible to perform meaning-preserving program transformations automatically [Backus 78]. Because functional languages have the property of referential transparency[6], it is possible to develop a rich algebra of programs, a feature that is an essential prerequisite for effective query optimization. The success of current relational database implementations relies in no small measure on the ability to optimize queries in the relational algebra, which is a (restricted) functional language.

Opportunities for Optimization Due to Parallelism: The presence of side-effects in a language introduces many read-before-write and write-before-read constraints. These constraints are artificial, and make it very difficult, if not impossible to move away from a purely sequential scheduling of evaluation activities. In functional languages, only the logically necessary data-dependencies remain; this reveals much fine-grained parallelism that permits great latitude in scheduling evaluation activity. This degree of freedom is exploited heavily in dataflow architectures to overcome memory latencies [Dennis,J.B.

[6]A modularity principle by which it is possible to treat a name and its definition as synonymous.

74, Arvind 78, Arvind 83a]; it seems likely that the same techniques could be used to overcome disk latencies too, a major factor in database system (in)efficiency.

The parallelism available in languages without side-effects also make them ideal candidates for high-performance multi-processor architectures; there are many projects attempting to exploit this idea, currently at the research stage [Arvind 83b, Dennis,J.B. 81a, Gurd,J.R. 85].

Rich Type Systems: Recently there has been much research into polymorphic type systems [Milner,R. 78]. Such type systems allow an economic style of programming *a la* Lisp/Scheme but with complete type safety and with the rich data abstraction facilities necessary for database work. These data abstraction facilities include user-defined abstract types, and type-heirarchies with inheritance[7].

The presence of updatable objects interferes with polymorphism in very non-modular ways. Updating a variable x may have the surprising (and unnerving!) effect of also changing the *type* of some other apparently unrelated variable **y**, merely because they each are members of two structures respectively that happen to share some third variable **z**. It greatly complicates life for the programmer to have to keep track of all sharing relationships in his program, however obscure. When there are no updatable objects, the issue of polymorphism can be separated cleanly from the issue of sharing.

Research into more expressive polymorphic type systems is very active [Burstall,R. 84, Cardelli,L. 84, Fairbairn,J. 85, MacQueen,D.B. 84, McCracken,N.J. 79, Mitchell,J.C. 84, Morris,J.H. 74, Martin-Lof,P. 73, Reynolds,J. 74, Reynolds,J. 83]; to our knowledge, all these efforts rely on the clean semantics of functional languages.

Polymorphic type systems in functional languages also appear to be useful in *interactive* database environments. These type systems are clean enough to allow building smart type-checkers capable of efficiently monitoring *incremental* changes in program modules during program development, due to editing, etc. [Nikhil,R.S. 85]. Thus there need be no distinction between environments for "query languages", used for quick one-off queries, and environments for "applications programming languages", used for large, complex queries and reports.

The presence of mutable objects also complicates abstract data type facilities. When building an object of abstract type, one is very careful to ensure that its internal

[7]Inheritance is sometimes called "Generalization" and/or "Specialization" in data model terminology.

representation cannot be accessed by procedures outside the abstract type definition. Such sharing can make it impossible to guarantee the invariants supposedly maintained by the data abstraction, because it is then possible to update the representation of an abstract object without using one of the "legal" procedures, and thus circumvent any integrity checks. This sharing comes about because an abstract object constructor often receives parameters from the outside which it may embed in the new abstract object that it creates. In functional languages, this sharing is safe and so the embedding may be avhieved very cheaply (by reference); in languages with side-effects, it may imply expensive and/or excessive *copying* of objects in order to ensure that representations of abstract objects are truly private. In addition to this execution overhead, there is the intellectual overhead for the programmer in that he has to be aware of this pitfall and must explicitly take precautionary measures.

4.3. Why Functional Databases?

Functional databases are databases that "never forget". From the functional point of view, a database is never modified; instead, it is "updated" by creating a *new* version that differs from the old in a manner dictated by the update operation. The versions form a (conceptually) infinite sequence of databases. Older versions are never deleted; they can be named and accessed just like the "latest" version.

There are many situations where this approach has already been taken, though perhaps not in such a formal sense. For example the notion of file versions in operating systems like Tops-20 and VMS is widely regarded as far superior to the no-backup or one-level-backup regimes of Unix and Tops-10. Another example is the "Undo" capability now appearing in some programming environments and text editors. Challis also poses arguments in favour of multi-version databases in [Challis,M.P. 82], where it is claimed that such databases simplify recovery and concurrency algorithms.

Many existing database systems *do* in fact retain all the information contained in previous versions of a database, but only for crash recovery and audit purposes-- the information is not directly accessible to the applications program.

A functional database offers a superior environment for concurrency control. Queries (read-only transactions) are never delayed, because it is always the case that there is available a latest, committed version of the database that is never going to be subsequently changed. Update transactions never delay-- and are never delayed by-- queries, because they always build new versions and thus never interfere with any version currently being examined by queries. Multiple updates, however, still must be serialized (though if two updates work on different parts of the database, they may actually be able to proceed in parallel).

The serialization of updates may also be easier in a functional setting. Two evaluation strategies commonly associated with the implementation of functional language implementations-- lazy evaluation [Friedman,D.P. 76, Henderson,P. 76] and early-completion [Dennis,J.B. 81a, Dennis,J.B. 81b]-- allow programs to report completion even though computations internal to the transaction are still suspended or in progress. This allows update transactions to "complete" earlier than they normally would, and in effect allow the *overlap* of multiple updates.

Functional databases permit reasoning about the history of states of a database, and because it is so easy to revert to an old version, to write "what-if" programs that experiment with possible futures. This kind of requirement seems an essential pre-requisite in so-called "intelligent" databases or "knowledge-bases".

Functional databases may also provide a richer environment in which to tackle the difficult problem of manually or automatically merging two separate databases into one. This problem arises in many situations-- e.g. when the operations of two or more companies are consolidated. Typically, the data in the two source databases will disagree in various ways. Having access to the histories of the two databases may alleviate the inconsistency problem because one may be able to use pattern-matching algorithms to establish consistent correspondences between the data at *different* points in the history of the two databases.

In a business setting there are many reasons why access to the history of the database is important. It may be a legal requirement to maintain records of all transactions, for example in a bank or personnel database (in fact banks already do this, but without any direct help from the database management system). The ability to examine time trends, for example in a stock-market or econometric modelling database, may also be an invaluable capability.

5. Implementation Issues

We are in the process of designing a prototype implementation of a Functional Database Programming Language. Here are some of our early design ideas.

There are many implementation techniques currently in use for functional languages.

We feel that the one that best suits our needs (especially for efficient support of lazy evaluation) is graph-reduction [Turner,D.A. 79, Johnsson,T. 84]. In this technique, programs are represented as directed graphs in a heap-structured memory. The internal nodes of the graphs are function-application and data-structure nodes, and the leaves are all constants. Evaluation of an expression involves repeatedly rewriting its graph until it is in normal form. The reader is invited to study the cited references for more details of graph-reduction.

To implement an FDBPL, the memory is organized into a sequence of heaps. Each version of the database is associated with its own heap, though this does not mean that the entire database version resides on that heap. Each update transaction initiates a new heap, and new objects constructed as part of that transaction are allocated in the new heap, though they may refer back to objects in older heaps. Thus an updated version of the database *shares* as much as possible of previous versions; only new objects necessarily allocated as part of the new version reside in the new heap for that version.

The programmer's model of the memory is that the entire sequence of databases (i.e. the sequence of heaps) is implemented in stable storage and is resilient across crashes. As a transparent optimization, pages of various heaps will have temporary working copies in high-speed, volatile memory. When a transaction is committed, the copy in stable storage is made to agree with any extant temporary copy.

A query (read-only) transaction by default first acquires the latest committed version of the database. From that point on, the entire query is evaluated with respect to that version (and older versions).

Because of finite storage capacity, it will in general be necessary to prune the sequence of databases at some version, and move all prior versions off-line. This is achieved with minor modifications to the same copying, compacting garbage-collector [Cohen,J. 81] used for normal storage management for the query and update languages. The garbage collector normally works by copying (and simultaneously compacting) all reachable data in a particular version "sideways" into a fresh heap for that version. On the other hand, when we prune the version history, all accessible data in older versions are copied "upwards" into a fresh heap at the oldest retained version. Decisions to prune the version history, and identification of the oldest version to be retained are taken dynamically depending on current storage utilization.

Functional semantics ensures that one cannot see "different" data as a result of this pruning activity. However when the database history is pruned, one will no longer be able to execute a query that interrogates a pruned version-- such attempts will invoke a run-time error (it is possible to determine statically whether a query *might* interrogate a pruned version, but one must run it to see whether it actually does so). Despite this, if the total storage capacity of the database system is subsequently increased (e.g. by buying more disk drives), it is possible easily and smoothly to re-integrate earlier pruned verions back into the version history.

We emphasize that despite the above discussion about finiteness of real databases, we believe our model is still a big step forward. We have relieved the database programmer of such concerns about finiteness in much the same way that the Algol programmer is relieved of certain finite storage allocation issues that bedevil the Fortran programmer, even though in a deep theoretical sense they are certainly equivalent.

It occurs to us that our model seems naturally suited to high-capacity write-once storage technology such as optical disks, but we have yet to study algorithms that make effective use of such devices in this functional setting.

A serious problem that needs to be investigated is the interaction of lazy evaluation with exception handling. We have already outlined various advantages of lazy evaluation in a database setting, including the ability to start a new update even before the current one has "completed". While this is desirable from a concurrency point of view, it raises new questions about how to handle exceptions. A query or update expression may force the evaluation of a computation that was suspended during some previous update. What is to be done if that suspended computation now raises an exception (such as a divide-by-zero?)[8].

We are exploring linguistic mechanisms for the meta-level "update" language. While it is necessary to make this language convenient to use, the main difficulties arise **a)** in ensuring that such meta-level environment-update operators maintain the type-correctness of the bindings used at the query-language level, and **b)** in allowing types to evolve incrementally as is inevitable in a long-lived database where one's model of the world changes over time. It is possible that recent work in safely packaging a certain level of dynamic type-checking in a generally statically type-checked system will be of use [Atkinson,M.P. 85, Cardelli,L. 85].

[8]This is related to the question of "cascaded aborts".

6. Conclusion

We have argued that it is essential that databases be treated as an integral part of programming languages rather than separately, as is common today. Further, we have claimed that the full range of type-structures in a programming language should be available for database objects. In this we are in substantial agreement with the PS-Algol (now Napier) and Galileo projects.

We then put forward the view that functional languages offer many advantages having to do with expressive power, rich type systems, interactive environments and optimization. We feel that databases should be viewed as environments in which queries (*i.e.* expressions) are evaluated, and that they should be "updated" by evaluating meta-level expressions that evaluate to new environments, thus creating a conceptually infinite sequence of database versions. We call such databases "Functional Databases". We argue that they are superior vehicles for concurrency control and that existence of a model of the history of a database is useful not only for recovery mechanisms but also in general for application programs.

We have begun developing a prototype functional database programming system to demonstrate the feasibility of this model. As part of this effort, we are studying language facilities to capture our model of update in the presence of a rich polymorphic type system with full type-checking.

References

[Albano,A. 83]
Albano,A., Cardelli,L. and Orsini,R.
Galileo: a Strongly Typed Interactive Conceptual Language.
Technical Report 83-11271-2, Bell Laboratories, 1983.

[Albano,A. 85a]
Albano,A., Giannotti,F., Orsini,R. and Pedreschi,D.
The Type System of Galileo.
In *Proc. Persistence and Data Types Workshop, Appin, Scotland.* August, 1985.

[Albano,A. 85b]
Albano,A., Ghelli,G., Orsini,R.
The Implementation of Galileo's Values Persistence.
In *Proc. Persistence and Data Types Workshop, Appin, Scotland.* August, 1985.

[Arvind 78]
Arvind,K.P., Gostelow and Plouffe,W.
An Asynchronous Programming Language and Computing Machine.
Technical Report 114a, Dept.of Information and Computer Science, University of California, Irvine, CA, December, 1978.

[Arvind 83a]
Arvind and Iannucci,R.A.
Two Fundamental Issues in Multiprocessing: the Dataflow Solution.
Technical Report TM 241, MIT Laboratory for Computer Science, September, 1983.
Revised July, 1985

[Arvind 83b]
Arvind, Culler,D.E., Iannucci,R.A., Kathail,V., Pingali,K. and Thomas,R.E.
The Tagged Token Dataflow Architecture.
Technical Report FLA, MIT Laboratory for Computer Science, August, 1983.
Revised October, 1984

[Atkinson,M.P. 81]
Atkinson,M.P., Chisholm,K.J. and Cockshott,W.P.
PS-Algol: An Algol with a Persistent Heap.
SIGPLAN Notices 17(7):24-31, July, 1981.

[Atkinson,M.P. 85]
Atkinson,M.P. and Morrison,R.
Types, Bindings and Parameters in a Persistent Environment.
In *Proc. Persistence and Data Types Workshop, Appin, Scotland.* August, 1985.

[Backus 78]
Backus,J.
Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs.
Communications of the ACM 21(8):613-641, August, 1978.

[Buneman 77]
Buneman,O.P. and Morgan,H.L.
Alerting Techniques for Database Systems.
In *Proceedings of the IEEE COMPSAC Conference, Chicago, Nov 1977.* IEEE, 1977.

[Burge 75]
Burge,W.H.
Recursive Programming Techniques.
Addison Wesley, Reading, Mass., 1975.

[Burstall,R. 84]
Burstall,R. and Lampson,B.
A Kernel Language for Abstract Data Types and Modules.
In *International Symposium on the Semantics of Data Types, Sophia-Antipolis, France (Springer-Verlag LNCS 173).* June, 1984.

[Cardelli,L. 83]
Cardelli,L.
ML Under Unix.
Technical Report, Bell Laboratories, 1983.

[Cardelli,L. 84]
Cardelli,L.
A Semantics of Multiple Inheritance.
In *International Symposium on the Semantics of Data Types, Sophia-Antipolis, France (Springer-Verlag LNCS 173).* June, 1984.

[Cardelli,L. 85]
Cardelli,L. and MacQueen,D.B.
Persistence and Type Abstraction.
In *Proc. Persistence and Data Types Workshop, Appin, Scotland.* August, 1985.

[Challis,M.P. 82]
Challis,M.P.
Version Management - or How To Implement Transactions Without a Recovery Log.
Database: Infotech State of the Art Report 9(8):435-458, January, 1982.

[Cohen,J. 81]
Cohen,J.
Garbage Collection of Linked Data Structures.
ACM Computing Surveys 13(3):341-367, September, 1981.

[Date,C.J. 75]
Date,C.J.
An Introduction to Database Systems.
Addison Wesley, Reading, Mass., 1975.

[Dennis,J.B. 74]
Dennis,J.B.
First Version of a Data Flow Procedure Language.
In Goos,G. and Hartmanis,J., editor, *Proc. Programming Symposium, Paris 1974,* Volume 19. Springer-Verlag, Berlin, 1974.

[Dennis,J.B. 81a]
Dennis,J.B.
Data Should Not Change: A Model for a Computer System.
Technical Report CSG Memo 209, MIT Laboratory for Computer Science, July, 1981.

[Dennis,J.B. 81b]
Dennis,J.B.
An Operational Semantics for a Language with Early Completion Data Structures.
In *Formal Descriptions of Programming Concepts*. Springer-Verlag, Berlin, 1981.

[Fairbairn,J. 85]
Fairbairn,J.
A New Type-Checker for a Functional Language.
In *Proc. Persistence and Data Types Workshop, Appin, Scotland, August 1985*. August, 1985.

[Friedman,D.P. 76]
Friedman,D.P. and Wise,D.S.
CONS should not evaluate its arguments.
In Michelson and Milner, editor, *Automata, Languages and Programming*, pages 257-284. Edinburgh University Press, Edinburgh, Scotland, 1976.
Also Tech. Rep. 44, Computer Science Dept., Indiana University, Bloomington, Indiana

[Gurd,J.R. 85]
Gurd,J.R., Kirkham,C.C. and Watson,I.
The Manchester Prototype Dataflow Computer.
Communications of the ACM 28(1):34-52, January, 1985.

[Henderson,P. 76]
Henderson,P. and Morris,J.H.
A Lazy Evaluator.
In *Proc. 3rd Symp. on Principles of Programming Languages, Atlanta*, pages 95-103. ACM, 1976.

[Johnsson,T. 84]
Johnsson,T.
Efficient Compilation of Lazy Evaluation.
SIGPLAN Notices 19(6):58-69, June, 1984.
Proc. ACM SIGPLAN '84 Symposium on Compiler Construction

[Liskov,B.H. 74]
Liskov,B.H. and Zilles,S.N.
Programming with Abstract Data Types.
SIGPLAN Notices 9(4):50-59, 1974.
(Proc. ACM SIGPLAN Conf. on VHLL)

[Lochovsky,F.H. 82]
Lochovsky,F.H. and Tsichritzis,D.C.
Data Models.
Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[MacQueen,D.B. 84]
MacQueen,D.B., Plotkin,G. and Sethi,R.
An ideal model for recursive polymorphic types.
In *11th ACM Symposium on Principles of Programming Languages, Salt Lake City, UT*. 1984.

[Martin-Lof,P. 73]
Martin-Lof,P.
An intuitionistic theory of types: predicative part.
North Holland, Amsterdam, 1973, pages 73-118.

[McCracken,N.J. 79]
McCracken,N.J.
An Investigation of a Programming Language with a Polymorphic Type Structure.
PhD thesis, Syracuse University, 1979.

[Milner,R. 78]
Milner,R.
A Theory of Type Polymorphism in Programming.
Journal of Computer and System Sciences 17:348-375, 1978.

[Mitchell,J.C. 84]
Mitchell,J.C.
Lambda Calculus Models of Typed Programming Languages.
PhD thesis, MIT Laboratory for Computer Science, 1984.

[Morris,J.H. 74]
Morris,J.H.
Towards More Flexible Type Systems.
In Goos and Hartmanis, editor, *Colloquium on Programming, Paris*, Volume 19. Springer-Verlag, Berlin, 1974.

[Nikhil,R.S. 84]
Nikhil,R.S.
An Incremental, Strongly-Typed Database Query Language.
PhD thesis, Moore School, University of Pennsylvania, Philadelphia, August, 1984.
Available as Technical Report MS-CIS-85-02

[Nikhil,R.S. 85]
Nikhil,R.S.
Practical Polymorphism.
In *Proc. Conf. on Functional Programming Languages and Computer Architecture, Nancy, France.* IFIPS, September, 1985.

[Reynolds,J. 74]
Reynolds,J.
Towards a Theory of Type Structure.
In Goos and Hartmanis, editor, *Colloquium on Programming, Paris 1974*, Volume 19, pages 408-425. Springer-Verlag, Berlin, 1974.

[Reynolds,J. 83]
Reynolds,J.
Types, abstraction and parametric polymorphism.
In *Information Processing 83.* North Holland, Amsterdam, 1983.

[Schmidt,J.W. 77]
Schmidt,J.W.
Some High Level Language Constructs for Data of Type Relation.
ACM Transactions on Database Systems 2(3):247-261, 1977.

[Turner,D.A. 79]
Turner,D.A.
A New Implementation Technique for Applicative Languages.
Software—Practice and Experience 9(1):31-49, 1979.

[Turner,D.A. 81]
Turner,D.A.
The Semantic Elegance of Applicative Languages.
In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*, pages 85-92. ACM, October, 1981.